Memoria - Práctica 2

Miguel Ángel Pérez Olivencia, Brandon René Morales Viracocha,
Gonzalo Morato de Tapia Caro, Ángela Poyatos Gómez

Desarrollo de Software

3º Ingeniería Informática - 2023/2024

Índice

1. Ejercicio grupal

Utilizar Flutter/Dart para crear el diseño adaptativo y perfectivo de la actividad 3 de la práctica 1, incluir la interfaz de usuario con widgets que permita al usuario interactuar con las diferentes funcionalidades implementadas.

1.1 Solución al ejercicio propuesto

Esta es una posible solución al ejercicio original de Python sobre la construcción de pizzas y bocatas utilizando el **patrón Builder**, adaptado para dispositivos móviles usando el framework Flutter y el lenguaje de programación Dart.

1.1.1 Clase MyApp y MyHomePage

La clase **MyApp** es la clase principal de la aplicación que hereda de *StatelessWidget*, esta clase representa la raíz del árbol de widgets de la aplicación. Al heredar de StatelessWidget no mantiene ningún estado interno haciendo que la clase sea más simple y eficiente.

En esta clase se establece la estructura básica de la aplicación, se define el título, tema y la página de inicio. MaterialApp es un widget propio de Flutter que crea la aplicación utilizando Material Design (diseño creado por google).

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Pizza & Bocata Builder',
    theme: ThemeData(
        primarySwatch: Colors.blue,
    ),
    home: const MyHomePage(title: 'Pizza & Bocata Builder'),
    );
  }
}
```

La clase MyHomePage también hereda de StatelessWidget y representa la página principal de la aplicación. Su constructor recibe el título de la aplicación que será el título de la página principal. El método build crea la interfaz de usuario, con una barra en la parte superior que muestra el título y en el cuerpo se van definiendo los distintos botones que el usuario puede seleccionar.

Cuando el cliente selecciona alguno de los dos botones se navega a otra página para decir el tipo especifico de *Bocata* o *Pizza* que corresponden a la clase *BocataOptionsPage* y *PizzaOptionsPage* respectivamente.

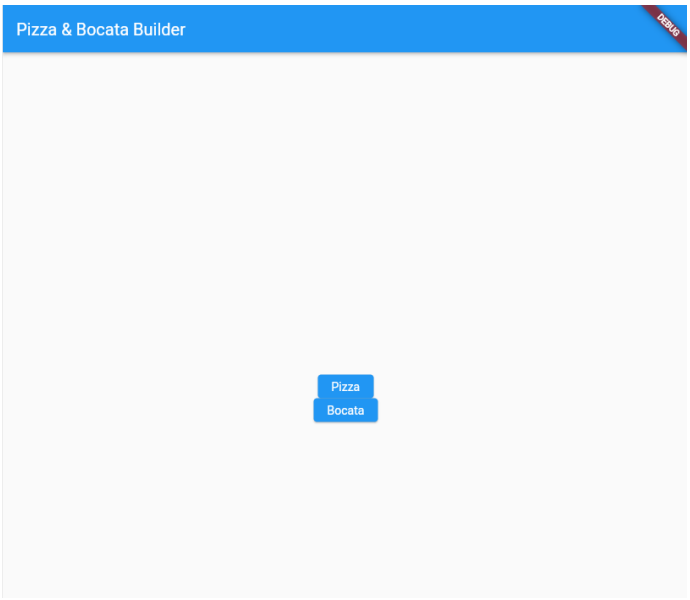

Ilustración 1. Pantalla de inicio de la aplicación.

```
class MyHomePage extends StatelessWidget {
  final String title;

  const MyHomePage({super.key, required this.title});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            ElevatedButton(
              //Cuando se pulse iriamos a la pagina pizzaoptionspage
              onPressed: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(builder: (context) => const PizzaOptionsPage()),
                );
              },
              child: const Text('Pizza'),
            ),
            ElevatedButton(
              onPressed: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(builder: (context) => const BocataOptionsPage()),
                );
              },
              child: const Text('Bocata'),
            ),
          ],
        ),
      ),
    );
  }
}
```

1.1.2 Clase PizzaOptionsPage y _MyPizzaOptionsState

Para no alargar la memoria se va a proceder a explicar el comportamiento de las clases relacionadas con la pizza.

Esta clase representa la página en la que los usuarios pueden elegir el tipo de pizza, esta clase hereda de *StatefulWidget* por lo que puede cambiar su estado a lo largo del tiempo. El estado interno de esta página es gestionado por el tipo **_MyPizzaOptionsState** y el método build se encarga de definir la interfaz.

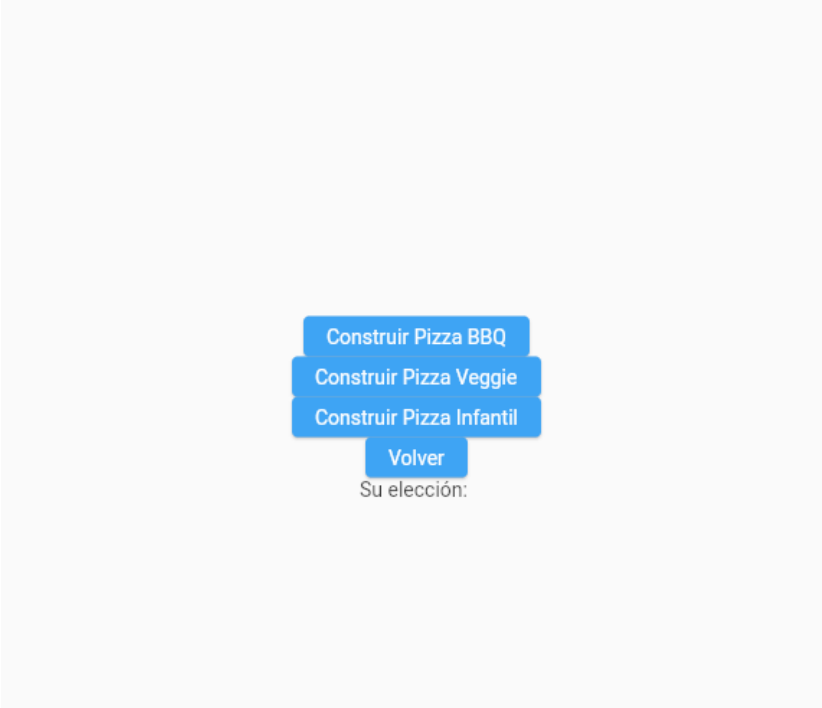

Ilustración 2. Opciones del producto.

```
class PizzaOptionsPage extends StatefulWidget {
  const PizzaOptionsPage({super.key});

  @override
  State<PizzaOptionsPage> createState() => _MyPizzaOptionsState();
}
class _MyPizzaOptionsState extends State<PizzaOptionsPage> {

  late Pizza pizza;
  late PizzaBuilder builder;
  String resultado = "";

  void construirPizza(){
    setState(() {
      pizza = builder.CreateNewPizza();
      resultado = pizza.str();
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Elegir Pizza'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
```

```
          children: <Widget>[
            ElevatedButton(
              onPressed: () {
                builder = BBQPizzaBuilder();
                construirPizza();
              },
              child: const Text('Construir Pizza BBQ'),
            ),
            ElevatedButton(
              onPressed: () {
                builder = VeggiePizzaBuilder();
                construirPizza();
              },
              child: const Text('Construir Pizza Veggie'),
            ),
            ElevatedButton(
              onPressed: () {
                builder = InfantilPizzaBuilder();
                construirPizza();
              },
              child: const Text('Construir Pizza Infantil'),
            ),
            ElevatedButton(
              onPressed: () {
                Navigator.pop(context);
              },
              child: const Text('Volver'),
            ),
            Text(
                '\nSu elección: $resultado',
            )
          ],
        ),
      ),
    );
  }
}
```

1.1.3 Clase Pizza

Esta clase al igual que en la práctica 1, representa una pizza y los atributos que puede tener (ingredientes, tipo de salsa, tipo de masa y tamaño). Tiene un método que devuelve las características de la pizza.

```
class Pizza{
  var ing = <String>[];
  late String salsa;
  late String tipoMasa;
  late String tamanio;

  String str(){
    return "\nIngredientes: $ing\nSalsa :$salsa\nTipo de masa :$tipoMasa\nTamaño :$tamanio";
  }
}
```

1.1.4 PizzaBuilder

La clase PizzaBuilder es una clase abstracta que tiene un método llamado **CreateNewPizza** que crea una instancia de la clase *Pizza* y proporciona métodos abstractos como AddIngredientes, TipoMasa, AddSalsa y

TamanioMasa. Estos métodos abstractos deben ser implementados por las subclases para definir cómo se construye cada tipo de pizza.

```
abstract class PizzaBuilder{
  Pizza pi = Pizza();
  Pizza CreateNewPizza(){
    AddIngredientes();
    TipoMasa();
    AddSalsa();
    TamanioMasa();

    return pi;
  }

  void AddIngredientes();
  void TipoMasa();
  void AddSalsa();
  void TamanioMasa();

}
```

1.1.5 Clases especificas BBQPizzaBuilder, VeggiePizzaBuilder y InfantilPizzaBuilder

Esta clase hereda de PizzaBuilder y proporciona una implementación específica para construir pizzas estilo barbacoa. En el método AddIngredientes, se añaden ingredientes, en TipoMasa, se define el tipo de masa y en AddSalsa, se especifica la salsa de barbacoa. Finalmente, en TamanioMasa, se establece el tamaño de la pizza como "mediana".

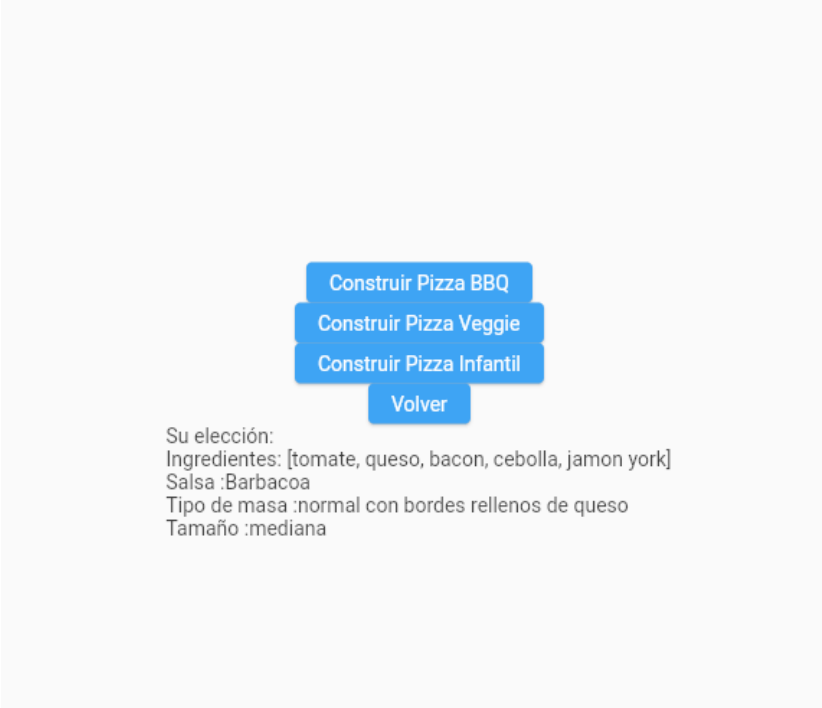

Ilustración 3. Producto seleccionado.

```
class BBQPizzaBuilder extends PizzaBuilder{
  @override
  void AddIngredientes(){
    pi.ing.add("tomate");
    pi.ing.add("queso");
    pi.ing.add("bacon");
```

```
    pi.ing.add("cebolla");
    pi.ing.add("jamon york");
  }

  @override
  void TipoMasa(){
    pi.tipoMasa = "normal con bordes rellenos de queso";
  }

  @override
  void AddSalsa(){
    pi.salsa = "Barbacoa";

  }

  @override
  void TamanioMasa(){
    pi.tamanio = "mediana";

  }
}
```

1.2 Mantenimiento

Al trasladar el ejercicio de Python a Flutter se ha tenido que modificar y adaptar el código para poder crear una interfaz de usuario y proporcionar la funcionalidad de la aplicación. Esta aplicación permite al usuario pedir un bocata o una pizza y, una vez seleccionado el producto, elegir el tipo concreto.

1.2.1 Mantenimiento adaptativo

El principal cambio realizado ha sido la reescritura de código, puesto que al utilizar Flutter, que sigue el patrón de diseño BLoC, se separa la lógica de la aplicación de la vista.

1.2.2 Mantenimiento perfectivo

En la práctica 1 no se había implementado una interfaz de usuario para la aplicación, por tanto, se ha introducido esta mejora. Flutter permite crear interfaces de usuario para aplicaciones y utilizando el patrón BLoC, se ha gestionado el estado de la aplicación mediante widgets. Aunque nuestra aplicación no tiene un gran tamaño, es una buena práctica para tener un buen rendimiento.

Por otro lado, el uso de Flutter proporciona compatibilidad con sistemas como iOS, Android y Web.

1.3 Widgets utilizados

La aplicación utiliza varios widgets de Flutter para construir la interfaz de usuario. Algunos de los widgets utilizados en la aplicación son los siguientes:

- **MaterialApp**: utilizado para definir la estructura básica de la aplicación. Proporciona funcionalidades esenciales como la gestión de rutas, manejo de temas, localización, etc.
- **Scaffold**: proporciona una estructura de diseño para la interfaz de la aplicación. Incluye una barra de aplicación (AppBar), un área de contenido (body), y opcionalmente un área de navegación.
- **AppBar**: muestra la barra de aplicación en la parte superior de la pantalla. Se utiliza para mostrar el título de la aplicación.

- **Center**: centra su hijo tanto vertical como horizontalmente en la pantalla.
- **Column**: organiza sus hijos en una columna vertical. Se utiliza para colocar varios widgets uno encima del otro.
- **ElevatedButton**: es un botón que muestra una elevación cuando se pulsa. Se utiliza para capturar la interacción del usuario.
- **Text**: muestra texto en la interfaz de usuario.
- **Navigator**: se utiliza para gestionar la navegación entre diferentes pantallas de la aplicación. Permite navegar hacia adelante y hacia atrás entre las páginas de la aplicación.
- **StatefulWidget y State**: se utilizan para manejar el estado mutable de la aplicación. StatefulWidget permite cambiar el estado durante la vida útil del widget, mientras que el State contiene el estado mutable y la lógica para cambiarlo.

1.4 Problemas encontrados

El principal problema encontrado ha sido la reactividad de la interfaz de usuario, es decir, que el estado de los distintos componentes se mantuviera y no se perdiera. Aunque una vez solventado este problema no ha sido complicada la resolución del ejercicio.